Computational Support for Multiplicity in Hierarchical Electronics Design

Richard Lin
richardlin@ucla.edu
University of California, Los Angeles
Los Angeles, USA

Rohit Ramesh
rkr@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Prabal Dutta
prabal@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Björn Hartmann
bjoern@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Ankur Mehta
mehtank@ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

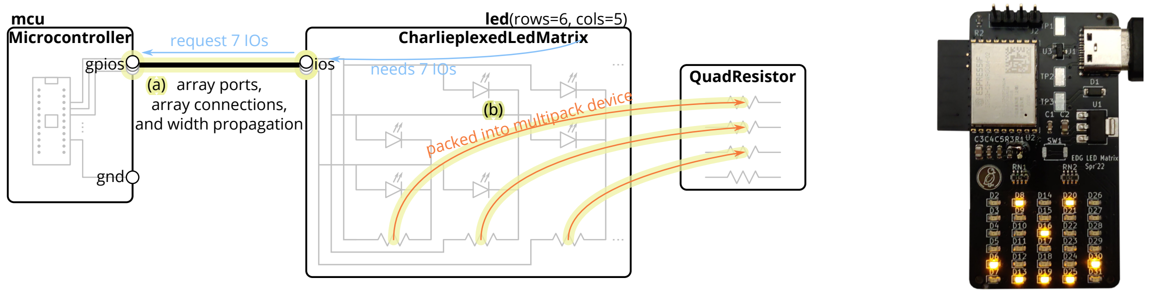

Figure 1: An example LED matrix circuit board that demonstrates our extensions to the standard blocks, ports, and connections hierarchical design model. The diagram on the left summarizes the user-facing design for this board – in particular, including the highlighted novel elements of (a) dynamically-sized port arrays (stacked ports) for the IOs on the LED matrix generator and the array connections (thick lines) to the microcontroller which also propagate width data (blue lines), and (b) multi-packing (orange lines) of the resistors in the LED matrix into a physical quad-pack resistor device.

ABSTRACT

While hierarchical design promises design process improvements through structures that better enable computational design, the basic model of blocks, ports, and connections lacks support for *multiplicity* – dealing with repeated instances of objects. In this work, we explore two extensions of that basic model to support two types of multiplicity, specifically in the context of board-level electronics where this is a common pattern. First, to support blocks that can be arbitrarily scalable across number of devices – e.g., an *n*-element LED array generator – we extend the existing fixed port interfaces for blocks with port arrays that can have dynamic width with automatic propagation through connections. Second, to support mapping abstract blocks in a design onto physical multipack devices – e.g., combining resistors across the design into a single quad-pack resistor device to optimize for fabrication – we introduce cross-hierarchy packing including support for shared pins. For both of these constructs, we describe the user-facing abstractions, internal representations, and compiler implementation, then demonstrate their end-to-end use through three example designs that we have fabricated and tested.

SCF '22, October 26–28, 2022, Seattle, WA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9872-5/22/10.
https://doi.org/10.1145/3559400.3561997

CCS CONCEPTS

• **Hardware** → *PCB design and layout*; *Hardware description languages and compilation.*

KEYWORDS

electronics design, hierarchical design, port arrays, multipack devices

ACM Reference Format:
Richard Lin, Rohit Ramesh, Prabal Dutta, Björn Hartmann, and Ankur Mehta. 2022. Computational Support for Multiplicity in Hierarchical Electronics Design. In *Symposium on Computational Fabrication (SCF '22), October 26–28, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3559400.3561997

1 INTRODUCTION

Hierarchical design is a common technique in many fields of engineering that helps manage complexity by working with higher-level and more intuitive building blocks. For example, in robotics this may be using a leg sub-assembly instead of discrete motors and struts, or in electronics this may be using a voltage converter subcircuit instead of a chip with inductors and capacitors around it. Furthermore, designers may draw upon libraries of these blocks, building upon the design work and expertise of others.

Yet, hierarchical design also describes a wide array of practices - from the process of sketching system-level diagrams to be manually translated into concrete designs, to tools that can work with these high-level descriptions and automatically generate fabrication-ready designs. This work focuses on extending hierarchical design tools in the context of board-level electronic circuit design. Our prior work [Lin et al. 2019] found that current mainstream design practices revolves around *schematic capture*, where users place symbols representing components onto a virtual schematic, then connect their pins together by drawing wires.

While mainstream schematic tools support notions of hierarchy by allowing components to represent a subcircuit, these primarily provide organization and readability benefits and are too basic for meaningful design automation. In particular, these subcircuits must be a single fixed instance, with components fully specified and allowing no parameterization or variability. While this encodes designs to the lowest level of detail and is necessary to produce a fabrication-ready schematic, this also limits the generalizability of any subcircuit. These component values and even circuit topologies are often tailored to one application and may be suboptimal or even incorrect in other contexts. For example, a voltage converter subcircuit encapsulates the controller chip and supporting resistors and capacitors into a single neat schematic block, but the resistor values also set the output voltage which limits the block to applications where that voltage is acceptable.

Recent work on more advanced tools extend this basic hierarchical structure with generators [Izraelevitz et al. 2017; Lin et al. 2020; SKiDL 2022], which define a subcircuit's implementation with code. Instead of being limited to a particular design instance with fully specified components, blocks can now encode a family of designs with the code resolving the details based on the desired application. This approach can cover a wide application space with a relatively small number of powerful generators. Continuing the voltage converter example, a generator could take in a target output voltage parameter and automatically choose resistor values to satisfy that specification.

However, *multiplicity* – which we define as dealing with repeated instances of objects in general – is a common pattern in electronics design where the above basic hierarchical structure falls short. One fairly direct form of multiplicity would be defining arrays of devices in a block – for example, an *n*-element LED array. Another form of multiplicity is a sort of inverse problem, packing and optimizing blocks in a hierarchical design into *multipack* devices that combine several devices into one physical component – for example, combining discrete resistors into a single physical quad-pack resistor array that offers size and cost advantages. We focus on the two above and common forms of multiplicity, though there are undoubtedly other forms that are out of scope of this work.

While all of the above are *possible* in any hierarchical design structure by essentially reducing down to schematics, doing so in a way that preserves the benefits of hierarchical design such as correctness-by-construction and reusability of blocks requires extensions to the fundamental hierarchical design model. For the array case, we would like to be able to define a single, reusable, width-parameterized block, but this requires the ports to scale with the device array. Furthermore, we would like its use to be correct-by-construction, automatically propagating and inferring width data across its connections. As for multipack, we would like to re-use general blocks that are built with individual components, but have those packed into these optimized devices even when the parts span across multiple blocks. We also want to preserve the correctness benefits of re-using proven libraries, even as their contents are modified to use these packed blocks.

In this work, we examine solutions to these issues through extensions of the basic hierarchical design model of blocks, ports, and connections. In particular, we contribute:

- **Port arrays**, to enable **dynamic interfaces** on blocks and specifically supporting width propagation, implicit width specification, and elastic requested connections to maximize automatic consistency across the design.
- **Packed devices** implemented with cross-hierarchy constructs, that enables these **cross-cutting optimizations** while preserving the abstraction and re-use benefits of a hierarchical design model.
- Implementation of the above constructs as an extension of our prior work on Polymorphic Blocks [Lin et al. 2020], a hierarchical board-level electronics HDL and compiler. The entire project is open sourced at https://github.com/BerkeleyHCI/PolymorphicBlocks.

Three example designs that we have fabricated – a charlieplexed LED matrix, a time-of-flight distance sensor array, and a multimeter – demonstrate real-world applications of these constructs, illustrate how they mesh with generator-based flows to increase meaningful design automation, and provide end-to-end demonstrations through the compiler to real circuit boards.

2 RELATED WORK

While hierarchical design applies for many fields of engineering, this work primarily focuses on and builds atop prior work on electronics design.

2.1 Hierarchical Design

Hierarchical design tools are the subject of active research in fields of engineering outside electronics design. For example, work in robotics include high-level specification of kinematic chains using libraries of parts [Desai et al. 2017] and integrated electronic and mechanical co-design of small robots [Mehta et al. 2015, 2014] including generation of foldable structures. In mechanical design more generally, similar ideas of hierarchical library-based design also appear, including as part of assembly-aware structural synthesis [Desai et al. 2018] and inferring parameterized templates from example designs [Schulz et al. 2014]. However, while these systems provide powerful examples of incorporating computational design with hierarchical design, such as by parameterizing on the length of drawers, they do not address multiplicity, such as by parameterizing on the number of drawers.

2.2 Electronics Design

The current mainstream approach of interactive graphical schematic capture dates back as far as the 1970s, as a replacement to punch card and text-based schematic entry [Matthews 1977]. Today, PCB suites are available as both commercial software [Altium 2018; Autodesk 2018; Mentor 2018] and open-source tools [KiCad 2018] and include the aforementioned schematic capture aspect as well as board layout. While earlier work explored bolting on hierarchical design for schematics as a separate tool [Bezzo et al. 2015], modern tools support basic hierarchical design by allowing components to represent subcircuits.

Schematic entry in modern tools tends to be a low-level task [Lin et al. 2019], often involving manual transcription from component datasheets. While design assistance tools exist, they are typically not integrated into the design suite and rely on the user to ensure design consistency. Examples of these tools include web apps that pick resistive dividers from a high-level ratio specification [Texas Instruments 2003] and create power conversion circuits [Texas Instruments 2022].

While modern schematic tools have features similar to array connections and for multipack devices, those approaches do not generalize to more automated and library-based flows. For arrays, schematic tools provide *busses* as a way to bundle multiple lines. However, the aggregation and disaggregation must be done manually, forcing the user to manually ensure design consistency. For multipack, users can place parts of a packed device as individual symbols. However, this may require modifications deep into the design hierarchy, which makes this incompatible with library-based flows where modifying the library is inadvisable.

Some novel commercial tools [Gumstix 2018; Sparkfun 2020] remix hierarchical schematic design with module-based design, where the tool allows users to design with high-level modules representing subcircuits (for example, for a microcontroller) instead of discrete components. However port arrays are not relevant as these tools do not support user-defined modules, and multipack devices are not relevant these tools do not allow inspecting into libraries for devices to pack.

Some recent research work on electronics design tools has examined synthesis approaches, where users specify a partial design and the system completes it through interface-driven synthesis. For example, a user might ask for sensors and motors, and the system inserts the missing power and compute systems by building outward from disconnected ports. Embedded Design Generation [Ramesh et al. 2017] translates and passes the circuit design problem to an SMT solver, while Echidna [Merrill et al. 2019; Merrill and Swanson 2019] uses heuristic-assisted tree search. However, fixed port definitions limit support for general blocks of scalable arrayed devices, while lack of support for multipack devices prevents the use of those more optimized components.

2.3 Electronics Hardware Description Languages

With hardware description languages (HDLs), users specify the circuit design as textual code instead of as graphical schematics. While this approach is not currently in common use for board design, recent work has examined how HDLs could support meaningful automation through programmatic circuit generation.

The simplest approach to a board HDL is schematics in textual form, such as with PHDL [Nelson et al. 2012]. More recent work [Bachrach et al. 2016; SKiDL 2022] extends these HDLs with generators, where users could define subcircuits with code and allow them be reactive to high-level parameters. For example, an LED-resistor circuit could automatically calculate the resistor value given a voltage and target current draw.

Our prior work on Polymorphic Blocks [Lin et al. 2020, 2021], the board-level HDL that this work builds on top of, further extends the generator concept by incorporating ideas from software engineering. In particular, a type system of blocks enables library builders to use generic and abstract superclass blocks, which can be *refined into* concrete subclass blocks by the system designer who would have more overall context. For example, an indicator LED block might use an abstract resistor, which then leaves the choice of surface-mount or through-hole resistor up to the system designer and allows the block itself to be widely applicable and directly used without modification.

Yet, all parts still must have fixed interfaces, so a parameterized length version of the above LED array block would not be possible. Furthermore, optimizations that cross hierarchical boundaries, like multipack devices, are not possible.

2.4 Digital Logic Design

Boards aside, hardware description languages are commonly in use for chip-level digital design, and array-typed ports are a common feature owing to the wide signal lines within chips. However, their semantics are more limited that what we propose and as a result require the designer to do more work to manually manage widths and consistency. Verilog, for example, requires explicit widths on array typed ports. Chisel [Izraelevitz et al. 2017], a newer generator HDL, does supports limited width propagation across module boundaries [Li et al. 2016], but generators cannot be reactive to widths of incoming connections.

Furthermore, arrays in digital logic HDLs are typically intended for signal buses where the lines are not interchangeable – for example, we would not want to swap the most significant bit and least significant bit. However, this is not necessarily the case in board-level electronics – where, for example, one IO pin on a microcontroller might be as good as the next – and new constructs like elastic port arrays which allow incoming connections without requiring a position make sense.

For chip design in general, while optimizations may also cross hierarchical boundaries, these are often handled in proprietary chip tools.

3 BACKGROUND

As this work extends Polymorphic Blocks [Lin et al. 2020], a novel HDL for board design, this section briefly recaps its underlying design model to provide context for the new work. Because this work focuses on design models rather than syntax, we present examples through diagrams of the block structure instead of user-facing HDL.

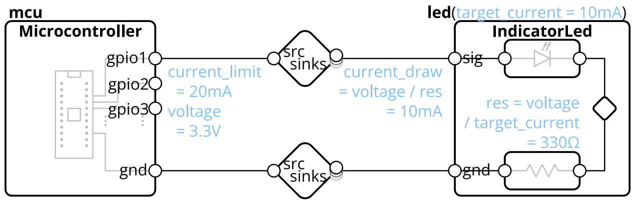

Figure 2: The internal design model with blocks (rectangles), ports (circles on those rectangles), and links (diamonds). Blocks are hierarchical and can contain other blocks and ports. Links mediate connections between ports. Parameters and expressions (in blue) can exist on all these elements and form the basis of the electronics model and design automation.

3.1 Hierarchical Block Model

Overall, users' textual HDL *elaborates* down to an internal model of blocks, ports, and links. Blocks represent subcircuits and components and contain ports on their boundaries, while ports represent pins and can be connected together through links. As the model is fully hierarchical, blocks can contain other blocks. Figure 2 shows an example of this model for a common embedded "hello, world" device, a microcontroller that controls an indicator LED, where the indicator LED is a subcircuit composed of internal discrete resistor and LED (sub)blocks.

The parameter and generator system enables meaningful design automation atop this basic structure. Blocks, ports, and links can define parameters (variables), relationships between them, and assertions on them. While these can contain arbitrary values, these typically represent electronic quantities like voltages and currents, and ratings like voltage and current limits. For example, this system can be used to automatically calculate the resistor value from the input voltage and target current, then check that the current draw is within the source's capabilities. Furthermore, generators allow arbitrary Python code to run and fill in a block's definition, for example the resistor block loading a parts table and filtering to find a suitable part.

The blocks can also be part of a type hierarchy. In this system, abstract blocks define a ports and parameters interface that subclass blocks can implement, and can then be replaced during compilation with concrete subclasses. This structure enables libraries of blocks that are appropriately general – for example, using any resistor instead of asking for an unnecessarily limiting specific part number – to allow that decision to be deferred until later – for example, by the system designer who opts for a machine-placed surface-mount board instead of a hand-soldered through-hole board. This also extends to different circuit topologies that perform the same function, such as a voltage converter that can be implemented by either a buck converter or a linear regulator.

3.2 Design Conventions

While the fundamental model does not enforce any dataflow order, our design convention for electronics has specifications flowing top-down and actual behavior flowing bottom-up. Continuing the voltage converter example, the enclosing block would specify a target output voltage for the converter (top-down dataflow), which would then choose parts to satisfy that specification. With parts chosen, the converter block then calculates the actual output voltage and makes that available to the enclosing block (bottom-up dataflow). This value can then be used as part of the specification for another block, for example in calculating the resistance needed for an LED (top-down dataflow again).

Our electronics model only models static behavior such as voltages, voltage limits, currents, and current limits. We do not model any notion of time, which would be in the domain of simulation, but we use ranges and interval arithmetic to provide static bounds on time-varying behavior.

3.3 Compiler

Overall, the compiler starts at the top-level block and recursively walks down the design tree to process blocks. On *compiling* each block, we process its connections, parameter assignments, and parameter assertions – for example, the automatic resistance calculation expression from above. A dependency graph manages relationships between parameters, generating a concrete value once all its dependency parameters have concrete values. This includes invoking generators when its input parameters are ready.

Once all blocks have been compiled, the design is complete. Then, a *board netlist* can be generated, containing a list of components and their connected pins. This can be directly imported into a board layout tool, in lieu of drawing a graphical schematic.

4 PORT ARRAYS

While the basic blocks, ports, and links structure with generators enables computational design through parameterization of variable values, the limitation of fixed port interfaces on blocks precludes parameterization of port counts. In particular, arraying devices – instantiating a set of n devices – is a common pattern in electronics, but this typically also requires the interface to scale with it. Without some way to dynamically size interfaces, we would be limited to manually defining a few static parameterizations, severely limiting the power of what could be computationally generated.

One example would be an n-LED array block, which requires one IO port for each LED. With blocks limited to fixed port interfaces, we would have to define a separate block for each width – a repetitive and inelegant solution. The situation is no better for the block user, where changing n would require changing the instantiated block type, then adding or deleting connections to be consistent with the new interface.

Ideally, we would like a single, reusable block that is parameterized by n, and for this sizing to automatically propagate to the block's interface and connections. This becomes more important with complicated devices having a non-trivial relation between their size parameter and the number of ports, where we would like computational tools to manage and encapsulate these relationships.

Our solution to this is *port arrays*, an interface supporting a parameterized number of ports. While this basic concept is common in digital logic HDLs like Verilog, their limited width propagation and connection semantics make them less than ideal for board-level electronics. Our implementation extends the connection semantics to include bidirectional width propagation to maximize automatic design consistency and correctness-by-construction, and elastic requested connections to better support common electronics design patterns.

4.1 Structure: Dynamicism in a Static Container

We structure a port array as a container port, which contains some parameterized number of internal *element* ports of the same type. This container port provides a static top-level interface for connecting to externally, and may be part of a supertype's interface definition. The inner ports then provide dynamic scaling within that structure.

The internal elements of a port array are named and always defined by the containing block. They may either be statically defined, or be defined by a generator to be reactive to parameters.

Using the *n*-LED array block as a running example, its static definition includes a port array of digital input lines, the width parameter, and the generator based on that parameter. During compilation, the generator can take the width value and dynamically instantiate that many elements in the port array.

4.2 Explicit and Implicit Specification

The LED array example above demonstrates an *explicit* width parameterization where the designer specifies a numeric width, which is typically the only way port arrays are defined in digital logic HDLs. However, this explicit width may be redundant with the number of incoming connections – connections to the port from the outside. In those cases, it can make more sense to derive an *implicit* width from those connections. This provides a potentially more natural input specification, eliminates a source of design inconsistency, and ultimately furthers correctness-by-construction.

Continuing the *n*-LED array running example, we may want to drive each LED from status outputs of a chip and therefore connect each status output to the *n*-LED array's input array. With this, we can infer the number of LEDs from the connections and avoid the need for an user-specified explicit count. Additional use cases include microcontrollers which present an elastic bag-of-IOs abstraction and *n*-to-one switches which are more naturally defined by connections instead of an explicit numeric width.

To allow for this, all port arrays have a *requested* parameter, which returns a list of incoming connections. These can be optionally named, which provides a mechanism to request specific ports, or ports with specific properties. For instance, a system designer might want to use a particular pin on a microcontroller to simplify physical layout. Like explicit widths, this requested parameter can also be a generator input, allowing the generator to automatically size the block.

The compiler matches requested names from the connection with element names of the port array. Where no requested name is specified, the compiler picks the next unused element. If a requested name matches no element names, the compiler flags an error.

As element names are defined by the block, it is up to the library writer to ensure correct element names, whether they are statically named or reactive to requested names. Static names may be useful for an *n*-to-one switch allocating sequential element names, while reactive names may be useful for an elastic-bag-of-IOs microcontroller to instantiate as many elements as needed for all incoming connections.

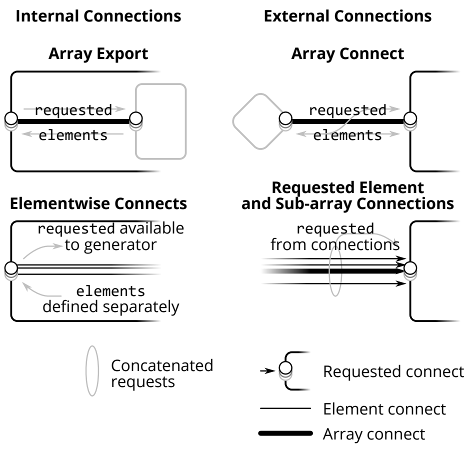

Figure 3: Summary of connection types and propagation rules. From internally, the port array can be connected as a whole to another array which propagates both requested and elements, or its elements can be defined and the element ports individually connected. From externally, the port array can also be connected as a whole via a link array which also propagates requested and elements, or individual connections can be made to requested subports and sub-arrays.

In general, requested flows inward (top-down, like specifications in our electronics design convention) and elements flows outward (bottom-up, like actual values in our convention).

4.3 Internal Connections

The connection semantics differ based on whether it is viewed from inside or outside the block. All the connection types and parameter propagation behavior are summarized in Figure 3.

From internally, the port array's elements can be explicitly defined, and each element port can be individually connected as if it were a typical top-level boundary port. Alternatively the port array can be connected as a whole array, exporting from an internal block's port array of the same type. This propagates requests inward and elements outward, and does not require a separate explicit definition of elements. These two connection types are mutually exclusive per port array.

4.4 External Connections

Viewed from externally, port arrays do not have defined elements, so elementwise connections must be done by requesting new ports, optionally with names. Newly requested ports can then be connected to like any other port on a block. There are no limits on requested ports, and as described above the compiler will try to match external requests with internally-defined elements.

Sub-arrays can also be requested from a port array, which allows port arrays to be connected into this port array alongside other array or element-wise connections. These sub-arrays have undefined elements, so elements must come from one of the connected port arrays. Unlike digital logic HDLs where connections to arrays must be at known indices, this structure of optionally-named-element,

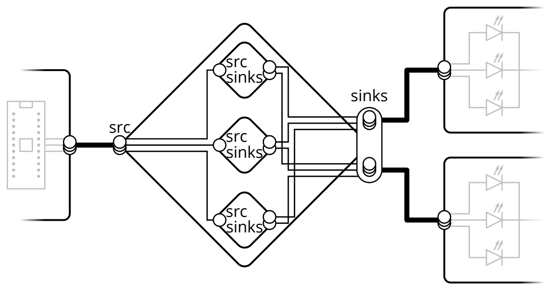

Figure 4: Example link array for a three element parallel connection of three digital IOs driving two LEDs each. The element link contains a single source port and an array of sink ports, while the encapsulating array contains an array of source ports and an array-of-array of sink ports.

elastic port arrays better suits board-level electronics where pins may be interchangeable and the specific element may not matter.

Similarly, port arrays can also be connected as an unit, either to other port arrays or to requested sub-arrays as described above. Aside from the above case where a port array is directly connected to a boundary port array on its enclosing block, only port arrays between sibling blocks (at the same level of hierarchy) may be connected[1].

Like single element connections where a link is required to mediate the connection, we introduce a *link array* construct that contains a parallel array of element links to mediate port array to port array connections. Link arrays are also parameterized by elements, and bidirectionally propagates elements between the link array and connected port arrays. While any port array may be a dataflow source or sink for elements, there must be exactly one source per link array, which then propagates to other connected port arrays. Figure 4 shows an example for a 3-element digital link array with one source and two sinks - such as would be the case for three microcontroller pins that each drives two LEDs.

Internally, the link array instantiates the element link type for each element, then connects the internal link's ports to the array's exterior ports. In general, the connections are of the form *link_port.elt* to *elt.link_port* for *elt* in elements if the element link port is a single port, or *link_port.i.elt* to *elt.link_port.i* if the element link port itself is an array (in which case the port on the link array is a nested port array).

Element-wise and sub-array requested connections and whole array connections are also mutually exclusive per port array.

4.5 Compiler Implementation

Compiler support for the port array construct includes defining and propagating the elements and requested parameters, expanding array connects into individual element-wise connects, and rewriting requests in connections to concrete element names.

As each block is compiled, the compiler gathers all connections to each port of each internal block. For port arrays, this may either be a direct array-to-array connection, in which case elements and requested are assigned according to the rules above, or this may be multiple connections to requested elements, in which case requested is assigned to be the concatenation of those requests. For element requests, this is the suggested name or an automatically generated name, while for sub-array requests, this is the elements of the sub-array prefixed by the suggested name of the sub-array.

For array-to-array connections, once the elements of the connection are known, they are expanded into single-element connects. Then, once the elements are known for a port array, such as after the block's generator runs, the requests of all connections can be matched with and rewritten as concrete element names. Afterwards, these look like, and are processed as, normal connections.

5 PACKED DEVICES

While port arrays enhance the abstraction power of blocks through variable-sized interfaces, in some cases it is helpful to break abstraction boundaries. In electronics, some devices are available *multipacked* – typically as multiple copies of the same device integrated into a single physical package. For example, a quadpack resistor contains four resistor *parts*, but requires less board area and costs less than using four discrete resistors.

While these multipack devices could be directly instantiated where they are used, baking this optimization into library blocks would be inelegant especially as not all applications may call for the same optimization and we may want to pack *part blocks* spanning across multiple blocks into a single multipack device. Furthermore, we would like to re-use unoptimized library blocks, while having a separate correct-by-construction packing process that preserves the proven nature of these library blocks.

This section describes our solution to the device packing problem, in a way that avoids the above pitfalls while largely staying consistent with and preserving the full power of the hierarchy blocks and generators model.

5.1 Internal Packed Device Model

Overall, the intuition for the internal design model is that each part block that is packed into a multipack device can be implemented by the multipack device itself. Model-wise, the multipack device would be inside each part block and export its ports and parameters into the part block. These part blocks can be thought of as thin wrappers around the multipack device, with the implementation of the part block delegated to the multipack device. This structure enables each part block to appear complete electronics-model-wise, while still having the multipack device be the ultimate source of truth.

However, the standard hierarchy block model cannot allow the multipack device to actually be within multiple separate part blocks, especially when these part blocks might also be scattered through the design hierarchy. To enable this, we introduce cross-hierarchy export connections and cross-hierarchy parameter assignments – *cross-hierarchy* meaning that these can "punch through" block

[1]In our hierarchical block model, when connecting more than one internal block's port to a boundary port, a *bridge block* is required to adapt between the boundary port and the link connecting the internal blocks. Intuitively, this is because a boundary port, viewed from inside, has flipped directionality – for example, a port that is a power sink externally looks like a power source internally. In concept, we can extend this to array connections by generating a bridge array, but this is not implemented. Users can still achieve the same goal more verbosely using element-wise connects, which will automatically instantiate the bridge blocks.

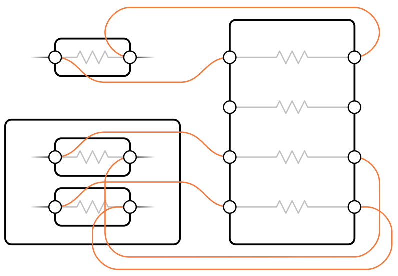

Figure 5: Example internal design model for a quad-pack resistor with part blocks delegating their implementation to an "internal" multipack block through cross-hierarchy connections. Packing is even supported when the part blocks are spread across the design hierarchy.

boundaries. Aside from not enforcing block boundaries, these have the same semantics as normal, non-cross-hierarchy export connections and parameter assignments, and no new compiler behavior is needed.

Figure 5 shows an example of the hierarchy model, where three resistors are packed into a quadpack device. Each resistor block that will be part of the quadpack is placed and externally connected as it normally would be. The multipack device defines four sets of resistor ports which is cross-hierarchy exported into resistor part blocks. The multipack device also defines a parameter for each part block's resistance spec, and an overall resistance. Cross-hierarchy assigns are used to propagate the resistance spec top-down from the part block to the multipack device, and propagate the selected resistance bottom-up back to the part block. Since packed resistor devices generally have the same resistance for all parts, the multipack device must select a resistor that satisfies all the part blocks' resistance specs.

Otherwise, the multipack device block itself looks and behaves like any other block. Multipack device block can support generators, which can depend on the values of these cross-hierarchy assigned parameters. For the multipack resistor, its generator selects parts from a parts table, similar to a single resistor.

5.2 Packing Specification

In addition to the functionality of a normal block, multipack blocks also must define packing rules. These specify how to translate user-facing packing directives into the internal model of cross-hierarchy exports and assigns.

This model, as in Figure 6, is the conventional view of a multipack device being comprised of multiple parts, and the inverse of the internal design model above. Here, the user declares virtual packed parts inside the multipack block, then defines virtual connections and assignments between the packed parts' ports and parameters and the multipack block's own.

The actual packing of specific blocks is left to the system designer and specified at the top level of the design. Here, the designer would instantiate the multipack blocks, then pack into its parts compatible blocks throughout the design tree. Like the refinements system, this specification of which blocks to pack where is fully manual, but the

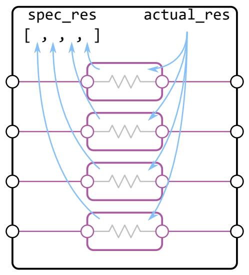

Figure 6: User-facing packing rules for the quad-pack resistor, which provides a more conventional view of packed devices. The purple boxes denote a packed part, while the purple and blue lines indicate how the cross-hierarchy exports and parameter assignments, respectively, generate between the packed part blocks and the multipack block.

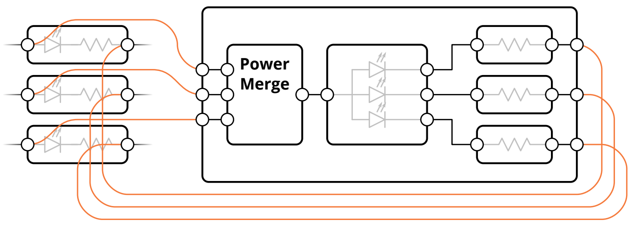

Figure 7: Example design model for a packed RGB LED using an internal merge block for the shared common-anode (positive voltage) pin that also checks that its three inputs are part of the same electrical connection.

translation from the user-facing packing directives into the internal model of cross-hierarchy exports and assigns is automated. This structure ensures that as long as the library's packing definition is correct, the system-level packing is correct-by-construction.

5.3 Packed Arrays

While the above describes individual virtual packed parts, virtual packed part arrays are also supported. The main difference is that instead of single-element virtual ports and connections, these have virtual port arrays and array connections. Similarly, the multipack device's boundary ports must be port arrays, and any parameters assigned from its parts must also be arrays. Since parts often have more than one port, the port array's element naming system ensures each part's ports are connected together[2].

The multipack resistors described above are actually implemented with packed arrays, allowing them to support an arbitrary number of packed resistors limited only by what is available in the parts tables.

5.4 Shared Pins

While all parts of the multipack resistor example have individual pins, packed devices may have shared pins – where ports on multiple parts must map onto the same pin on the multipack device. For example, a red-green-blue (RGB) LED can be defined as a tri-pack of three individual LEDs, but the device itself would have a single, common power pin. In these cases, the multipack block must still define independent pins for each packed part, but it can map those internally to a shared-pin device.

The mapping can involve a merge block, which takes multiple pins on the input and produces one pin on the output. This could include an assertion to check that all the input ports are part of the same electrical connection, then electrically connect them to the output port. A power merge, as for the RGB LED example and as shown in Figure 7, would propagate the voltage directly but evenly split the current draw among the three inputs, and check that all three inputs are part of the same electrical connection.

6 EXAMPLES

To demonstrate and end-to-end validate these constructs and their implementation, we designed three example devices and physically built and tested them. Each of these showcases a different use of port arrays and multipack devices.

Each example has the schematic-equivalent fully built in our HDL and compiled down to a netlist. Since our work does not cover layout, we imported the netlist into KiCad and manually placed and routed the board. Pin assignments and packing assignments were manually specified to simplify the layout.

The system also generates part numbers for each component, which provides the data for the board to be assembled by the factory. Some common components such as resistors, resistors arrays, and capacitors were chosen automatically from a factory-provided parts table by automatically matching against specifications in the design such as resistance and capacitance. Other components, such as the choice of microcontroller, were manually specified.

6.1 Charlieplexing LED Matrix

The charlieplexing LED matrix board, as shown in Figure 1 and with internal model in Figure 8 is a scaled-up and more complex version of the basic LED array running example. It drives a five-by-six grid of LEDs using just seven IO pins on a relatively low pin count microcontroller, an ESP32-C3 module. The microcontroller is WiFi capable and serves up a basic web page allowing the user to toggle between LED patterns.

The LED matrix itself uses a charlieplexing circuit, which in short allows each IO pin to drive both a row and a column, making the IO requirement roughly the maximum of the number of rows or columns, plus one. With ports arrays, the charlieplexed LED matrix is a block that takes in a row and column count, generates the internal charlieplexed circuit topology, and produces an appropriately-sized signal array. This is connected to the microcontroller IOs as an array, with the compiler requesting as many pins on the microcontroller as needed.

[2]There currently is no support for linking a parameter array's elements with the corresponding part – parameter arrays are unordered. However, this could be done with dict-typed parameters.

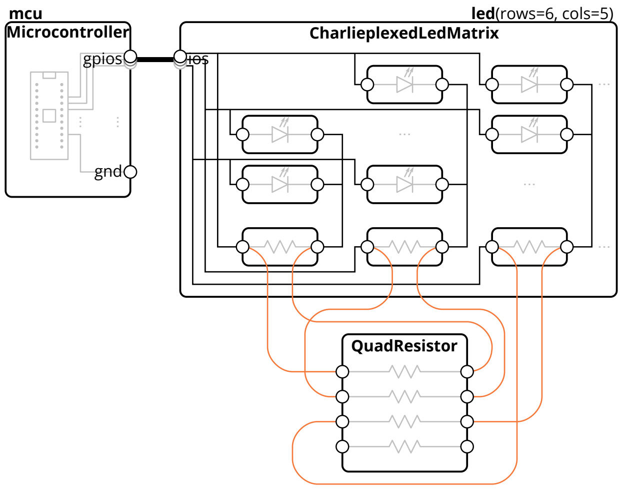

Figure 8: The fully expanded internal model for the charlieplexed LED array example from Figure 1, including the port array interface that enables the arbitrary-size charlieplexing LED matrix block and the multipacking as cross-hierarchy exports.

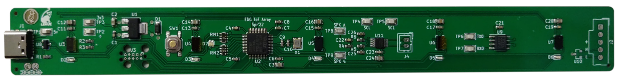

Figure 9: Example distance sensor array board. Port arrays enable the distance sensor array by exposing the shutdown pin for each individual device, while multipacking combines three discrete LEDs into an RGB LED.

Overall, the charlieplexing block encapsulates the knowledge needed to drive a large number of LEDs with just a few pins. The user only needs to specify the row and column count and connect the signal lines, without needing to understand the details of the internal circuit. Furthermore, as widths are automatically propagated, this also frees the user from manually calculating the pin counts to keep the design consistent.

This charlieplexing topology only uses one resistor per column, so it creates five resistors. A top-level packing rule replaces those discrete resistors with the resistor arrays on the board, which are much more compact. Both are partially packed, with three resistors on the left device and two on the right device being used.

The design also uses an abstract microcontroller, defined as a power supply and a set of ports arrays of common IO types like digital, SPI, I2C, and USB. An user-specified refinement picks the concrete ESP32-C3, which maps the generic IO arrays to the chip's pins.

6.2 Distance Sensor Array

The second example in Figure 9 is an array of distance sensors, using the VL53L0X time-of-flight laser ranging devices. While this board just has a linear array of sensors for simplicity, real-world applications might put these on a more expensive flexible PCB for wrap-around 360-degree distance sensing.

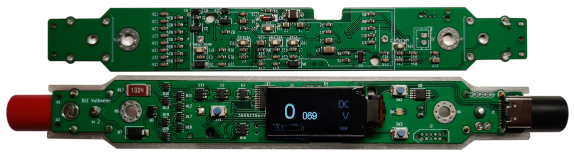

Figure 10: Example BLE multimeter board. Port arrays are crucial to the arbitrary-width mux tree generator and array connections on its control line encapsulate and hide the sizing details from the user.

The interesting aspect of these sensors is that they communicate by I2C but all devices share the same address, making it not straightforward to use multiple of these devices together. Instead, the shutdown pin of each device must be used to select the active device. To achieve this, the sensor array block defines both an I2C port shared by all the devices and a shutdown signal array that is connected to each individual device. Similar to the charlieplexing example, this encapsulates this implementation detail into an easy-to-use block. Without port arrays, it would be impossible to encapsulate the array of shutdown pins, forcing users to instantiate the sensors directly and connect each shutdown pin individually.

This also instantiates the simple LED array from the running examples, one LED for each distance sensor plus three more multipacked into an RGB LED. The RGB LED is a common-anode device (the positive connections of each LED are tied together into a single pin on the device), and provides an end-to-end example in hardware of shared pins in multipack devices.

This design similarly uses an abstract microcontroller, but with an STM32 refinement. Unlike the other examples which use microcontroller modules, this places the discrete microcontroller with supporting components like capacitors and crystals directly on the board.

6.3 BLE Multimeter

Finally, the Bluetooth Low Energy (BLE) multimeter in Figure 10 demonstrates a more complete and complex example. This uses an nRF52840 BLE module as the microcontroller, various power supply circuits including the 1.5 to 5 volt boost converter, discrete 24-bit analog-to-digital converter (ADC), and signal conditioning circuits. Bluetooth allows this device to be a compact field instrument by enabling optional connectivity to a smartphone with a larger display.

Of note is the variable resistor divider, which scales down the input voltage to the 2.4-volt range the ADC tolerates. For example, for a 10 volt input, the 100k resistor is selected for the bottom of the divider, which along with the 1M input resistor forms an approximately 10:1 divider that gives around 1 volt to the ADC. This is implemented using analog multiplexers (muxes), devices that select which of several inputs is connected to the output based on a control signal. In particular, we use an analog mux tree generator: as we define four ranges for the variable divider but prefer two-to-one analog mux devices, this generates three discrete analog muxes to implement a four-to-one mux.

This generator uses port arrays for both the input lines and control signals. The width is implicitly parameterized by the number of connected input lines, and the number of control signals is the log2 of the width. The control signals are array-connected to the microcontroller, and the compiler requests as many pins on the microcontroller as needed. Port arrays are critical to enabling this arbitrary-width mux generator which requires arbitrary-width ports, as well as completely encapsulating the control port width calculation. In general, this is another example of a non-trivial circuit generator enabled by port arrays, with non-trivial width relationships between ports.

Unlike the ESP32-C3 and STM32, the nRF52 requires both a 5v and 3.3v supply when using USB, so this cannot be implemented by the abstract microcontroller block which allows only one power supply port. Instead, the nRF52840 block is directly used in the design and both of its power inputs connected.

Since low-cost assembly services are limited to a single side of components, only the bottom side was assembled by the factory. The top side was manually soldered.

7 DISCUSSION AND FUTURE WORK

As shown by the examples, port arrays generalizes the block model from fixed ports to arbitrary-width ports, and this enables blocks to define and encapsulate a wide class of subcircuits that are parameterized width, even with complex internal topologies. Furthermore, multipack devices enable a form of cross-hierarchy optimization that helps bridge the ideal models for computational design tools with the messy nature of real-world systems.

Both these constructs combined help push the vision of Polymorphic Blocks, where lower-level design details can be encapsulated and computationally designed with powerful generator blocks. This, in turn, pushes the level of design abstraction higher, towards the system architecture level that captures the essence without getting mired in the details and is friendlier to novices.

7.1 Extending the Design Model

While this work extends the hierarchy model in a powerful way, the examples show that there is more to be done. As in the BLE multimeter example, the microcontroller also needed a 5-volt input and could not fit the abstract microcontroller interface. This pattern of almost-fits-an-interface-but-not-quite appears throughout electronics. For example, a voltage converter might fit the power-in, power-out, ground interface, but plus a shutdown pin.

A simple approach that works with the current interface is to use adapter blocks to simplify a complicated interface down to a more basic interface. However, this potentially risks needing a lot of adapter blocks which would both clutter the library and make evaluation of alternatives harder. Additional model extensions may help more directly encode this use case, including adapting more programming languages concepts like mixins.

Furthermore, while multipacking captures one particular type of cross-hierarchy optimization, electronics is filled with highly-integrated devices that don't perform just one neat function. For example, microcontrollers often have more than only IOs: they may integrate onboard power conversion or analog filtering circuits. While these can be shoehorned into the multipacking abstraction and the underlying cross-hierarchy constructs, perhaps there are other extensions that address this class of problems cleaner.

7.2 Verification

While generators are great at producing many designs, whether those designs work is another question completely. The electronics model does act as a first-order check, but trades detail for speed and usability.

Case in point: the LED matrix example actually has an error. Resistor R1 was mistakenly specified as a pull-down resistor when it needed to be a pull-up resistor. As a result, the microcontroller was unable to take new code until fixed with the little wire in Figure 1 next to R3 and R1.

One benefit of a library-based approach is that, ideally, these mistakes would be made only once before being fixed in the library forever. However, it would be worth exploring how tools could improve the process, either through stronger modeling and simulation, or just sharing information between all those who have built a device.

7.3 Opportunities for Design Space Exploration

One major benefit of port arrays is that it increases the abstraction power of blocks – for example, two of the examples used abstract microcontrollers, and a refinement specification replaces them with a concrete microcontroller subcircuit without requiring design changes. Somewhat abstractly, this abstract base block structure actually encodes a design space: all the subclasses are candidates for refinement.

While the choice of refinements is currently fully manual, future work could automatically explore this design space. This may come in the form of automatically making a best choice, a more human-in-the-loop approach presenting the user with trade-offs between different options, or some mix of the above that allows finer-grained control on top of an automatic optimizer.

While the block class hierarchy narrows down the design space compared to a general synthesis problem, a typical design still contains a large number of abstract parts and results in a high-dimensional problem. Human control over the search space can also help reduce the computational power needed by applying human intuition – for example, while a typical design might have a ton of abstract capacitors, a human designer may realize they should be all of the same subclass instead of searching through their combinatorial explosion. This process could not only try to optimize for currently modeled electrical parameters like current draw, but might also take into account factors like cost, component count, and even fabrication processes - for example, preferring larger components for hand-soldering or smaller components for machine assembly.

7.4 Other Domains

Finally, the hierarchical model is not unique to electronics: there is much related work across other domains. The constructs we described here and the more general idea of multiplicity may be just as applicable elsewhere, for example a parameterized truss generator that has a variable number of attachment points based on its length, or modeling an off-the-shelf robot arm that also includes a camera at the end.

8 CONCLUSION

While hierarchical design is the subject of active research and promises a more efficient and less error-prone design process, the basic blocks, ports, and connections model can be limiting. In this work, we expand that fundamental model with port arrays to better support parameterized blocks that also need parameterized interfaces, and with multipack devices to better support that kind of cross-hierarchy optimization. Our examples provide end-to-end demonstrations that these do work in a board-level electronics design context, and can improve the abstraction and encapsulation capabilities of these hierarchical design tools.

In the larger picture, the hope is that these constructs can be another part of the puzzle to bridge theory and practice for these advanced design tools, enabling them to be useful for a wide variety of real-world problems and bringing their benefits to a wider group of people.

ACKNOWLEDGMENTS

This work was supported in part by DARPA grants HR00112110008 and FA8750-20-C-0156 (SDCPS). The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, nor does this imply any official endorsement. Approved for public release; distribution is unlimited.

REFERENCES

Altium. 2018. *Altium Designer.* https://www.altium.com/altium-designer/

Autodesk. 2018. *EAGLE | PCB Design Software.* https://www.autodesk.com/products/eagle/overview

Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W Haldane, and Richard Lin. 2016. JITPCB. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on.* IEEE, 2230–2236. https://doi.org/10.1109/IROS.2016.7759349

Nicola Bezzo, Peter Gebhard, Insup Lee, Matthew Piccoli, Vijay Kumar, and Mark Yim. 2015. Rapid Co-Design of Electro-Mechanical Specifications for Robotic Systems *(International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Vol. Volume 9: 2015 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications).* https://doi.org/10.1115/DETC2015-47472 arXiv:https://asmedigitalcollection.asme.org/IDETC-CIE/proceedings-pdf/IDETC-CIE2015/57199/V009T07A009/4225178/v009t07a009-detc2015-47472.pdf V009T07A009.

Ruta Desai, James McCann, and Stelian Coros. 2018. Assembly-Aware Design of Printable Electromechanical Devices. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18).* Association for Computing Machinery, New York, NY, USA, 457–472. https://doi.org/10.1145/3242587.3242655

Ruta Desai, Ye Yuan, and Stelian Coros. 2017. Computational abstractions for interactive design of robotic devices. In *2017 IEEE International Conference on Robotics and Automation (ICRA).* 1196–1203. https://doi.org/10.1109/ICRA.2017.7989143

Gumstix. 2018. *Geppetto.* www.gumstix.com/geppetto/

A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

KiCad. 2018. *KiCad EDA.* http://kicad-pcb.org/

Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. 2016. Specification for the FIRRTL Language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9* (2016).

Richard Lin, Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. 2020. Polymorphic Blocks: Unifying High-Level Specification and Low-Level Control for Circuit Board Design. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA)

(UIST '20). Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/3379337.3415860

Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. 2019. Beyond Schematic Capture: Meaningful Abstractions for Better Electronics Design Tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 283, 13 pages. https://doi.org/10.1145/3290605.3300513

Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Bjoern Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1039–1049. https://doi.org/10.1145/3472749.3474804

Andrew J. Matthews. 1977. A Human Engineered PCB Design System. In *Proceedings of the 14th Design Automation Conference (DAC '77)*. IEEE Press, Piscataway, NJ, USA, 182–186. http://dl.acm.org/citation.cfm?id=800262.809124

Ankur Mehta, Joseph DelPreto, and Daniela Rus. 2015. Integrated Codesign of Printable Robots. *Journal of Mechanisms and Robotics* 7, 2 (05 2015). https://doi.org/10.1115/1.4029496 arXiv:https://asmedigitalcollection.asme.org/mechanismsrobotics/article-pdf/7/2/021015/6253002/jmr_007_02_021015.pdf 021015.

Ankur M Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. 2014. Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2892–2897.

Mentor. 2018. *Xpedition Enterprise*. https://www.mentor.com/pcb/xpedition/

Devon J. Merrill, Jorge Garza, and Steven Swanson. 2019. Echidna: Mixed-Domain Computational Implementation via Decision Trees. In *Proceedings of the ACM Symposium on Computational Fabrication* (Pittsburgh, Pennsylvania) *(SCF '19)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3328939.3329004

Devon J. Merrill and Steven Swanson. 2019. Reducing Instructor Workload in an Introductory Robotics Course via Computational Design. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 592–598. https://doi.org/10.1145/3287324.3287506

Brant Nelson, Brad Riching, and Josh Mangelson. 2012. Using a Custom-Built HDL for Printed Circuit Board Design Capture. PCB West 2012 Presentation.

Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. 2017. Turning Coders into Makers: The Promise of Embedded Design Generation. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication* (Cambridge, Massachusetts) *(SCF '17)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3083157.3083159

Adriana Schulz, Ariel Shamir, David I. W. Levin, Pitchaya Sitthi-amorn, and Wojciech Matusik. 2014. Design and Fabrication by Example. *ACM Trans. Graph.* 33, 4, Article 62 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601127

SKiDL. 2022. *SKiDL*. https://github.com/devbisme/skidl

Sparkfun. 2020. *À La Carte*. https://alc.sparkfun.com/

Texas Instruments. 2003. *Voltage Divider Calculator*. https://www.ti.com/download/kbase/volt/volt_div3.htm

Texas Instruments. 2022. *WEBENCH® Power Designer*. https://www.ti.com/design-resources/design-tools-simulation/webench-power-designer.html